Graphs: Undirected Graphs

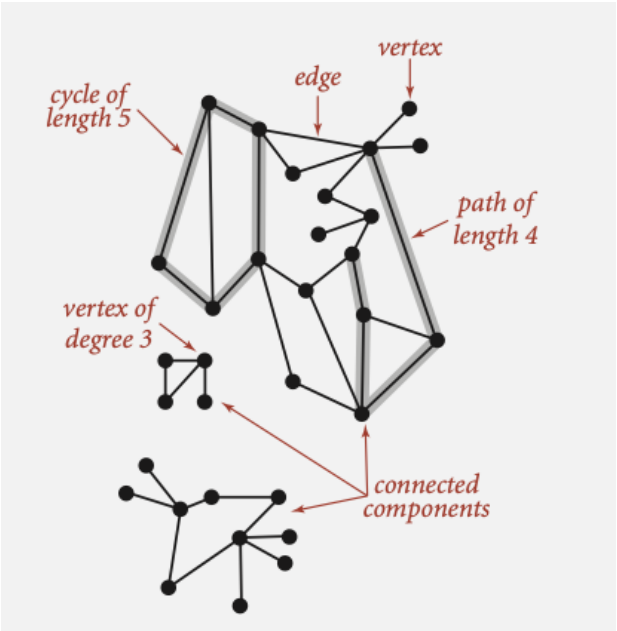
Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 4.1) and Prof. Janicki's course slides

Graph terminology

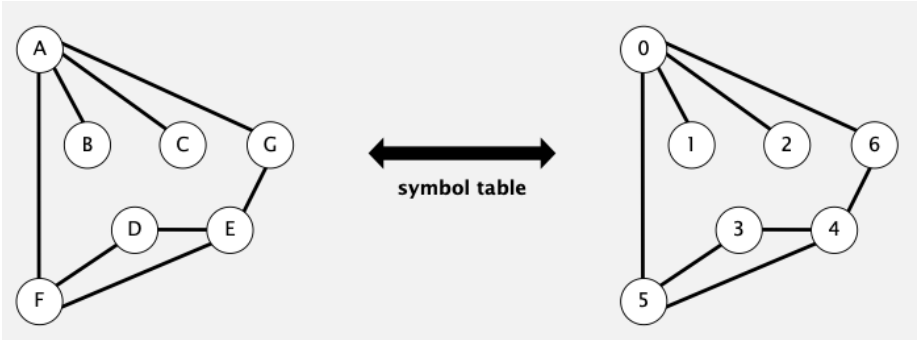
- Path. Sequence of vertices connected by edges.
- Cycle. Path whose first and last vertices are the same.
- Two vertices are connected if there is a path between them.
- If an edge exists between two vertices, they are adjacent to each other, and the edge is incident to both vertices.
- The degree of a vertex is the number of edges incident to it.



Graph representation

Vertex representation The book uses 0 to V-1 to label V vertices.

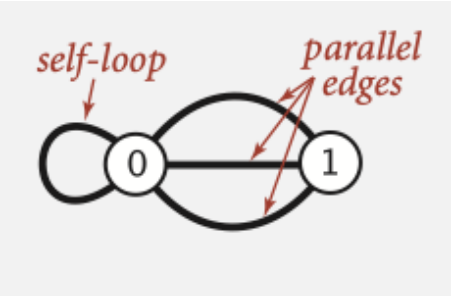
Maintain a symbol table to convert between names and integers.



The definition in the book allows two anomalies:

A self-loop is an edge that connects a vertex to itself.

Two edges that connect the same pair of vertices are parallel.



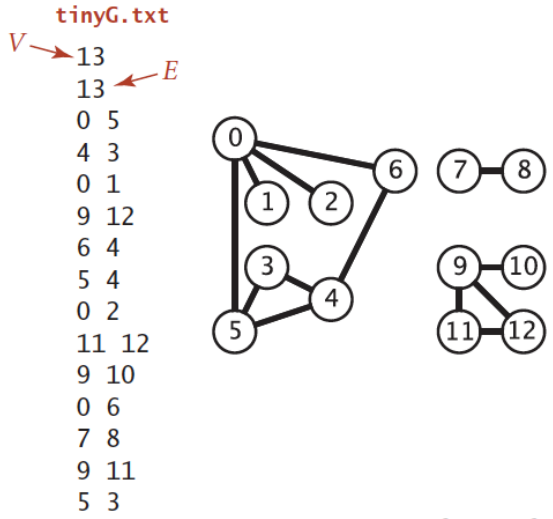
In implementations the book allows self-loops and parallel edges, but they are not included in examples.

Therefore, they refer to every edge just by naming the two vertices it connects.

Graph API

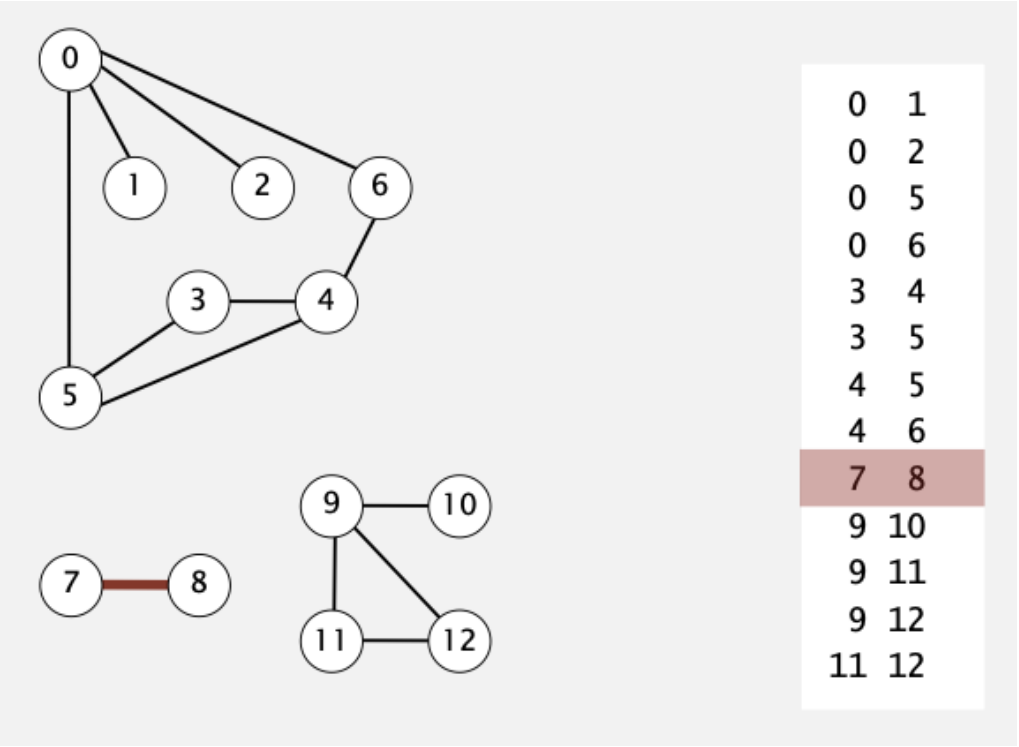
public class Graph	
Graph(int V)	*create an empty graph with V vertices*
Graph(In in)	*create a graph from input stream*
void addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer> adj(int v)	*vertices adjacent to v*
int V()	*number of vertices*
int E()	*number of edges*

Graph(In in) assumes an input format consisting of 2E + 2 integer values: V, and E, then E pairs of values between 0 and V-1, each pair denoting an edge.



Set-of-edges graph representation

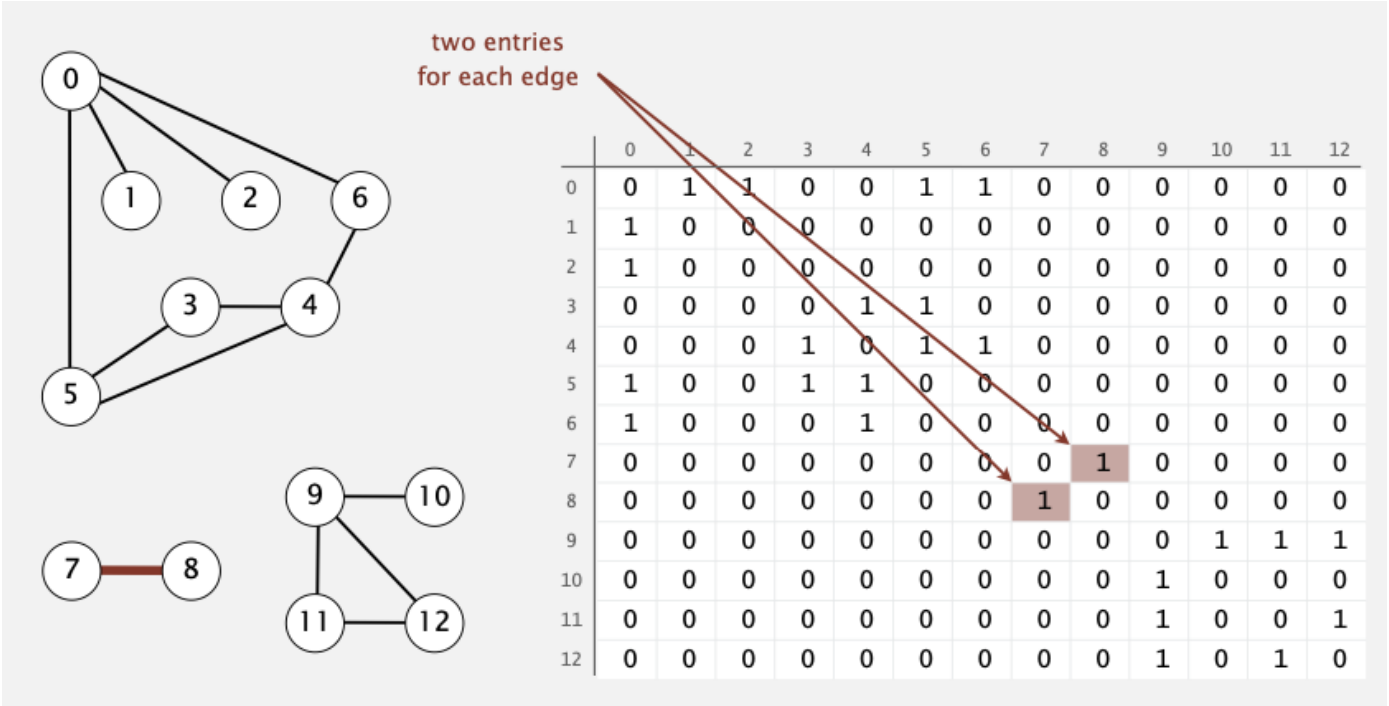
Maintain a list of the edges (linked list or array).



Issues: Basic operations are not performed efficiently. For instance, adj().

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;
for each edge v-w in graph: $adj[v][w] = adj[w][v] = true$.

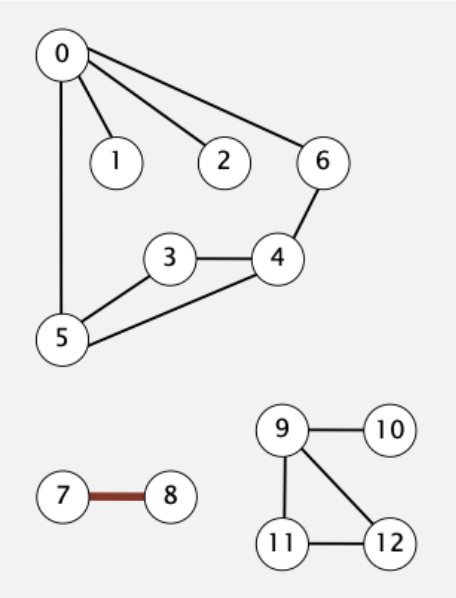


	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

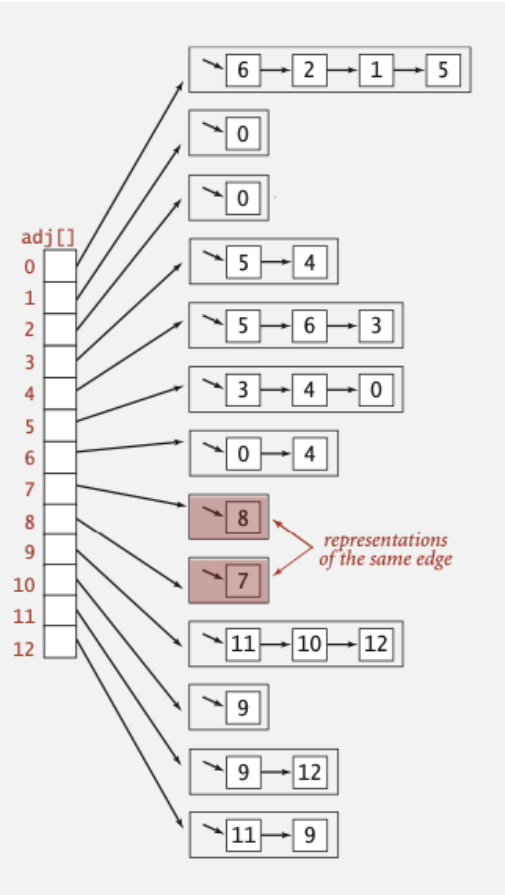
Issue: Need enormous amount of space to represent large graphs

Adjacency-list graph representation

Maintain vertex-indexed array of lists.



Issue: Cannot represent parallel edges.



In practice. Use adjacency-lists representation, as real-world graphs tend to be sparse [huge number of vertices, small average vertex degree].

Adjacency-list graph representation: Java implementation

Bag implementation of the adjacency list allows for parallel and self-loops.

```
public class Graph
{
   private final int V;
   private Bag<Integer>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists (using Bag data type)

create empty graph with V vertices

add edge v-w (parallel edges and self-loops allowed)

iterator for vertices adjacent to v

Search API

public class	Search	
	Search(Graph G, int s)	*find vertices connected to a source vertex s*
boolean	marked(int v)	*is v connected to s?*
int	count()	*how many vertices are connected to s?*

Graph-processing API (warmup)

- The Search API can be implemented using the the union-find algorithms.
- We will study a better implementation based on depth-first search – a fundamental recursive method that follows the graph's edges to find the vertices connected to the source.

Depth First Search

Goal. Systematically traverse a graph.

DFS (to visit a vertex v)

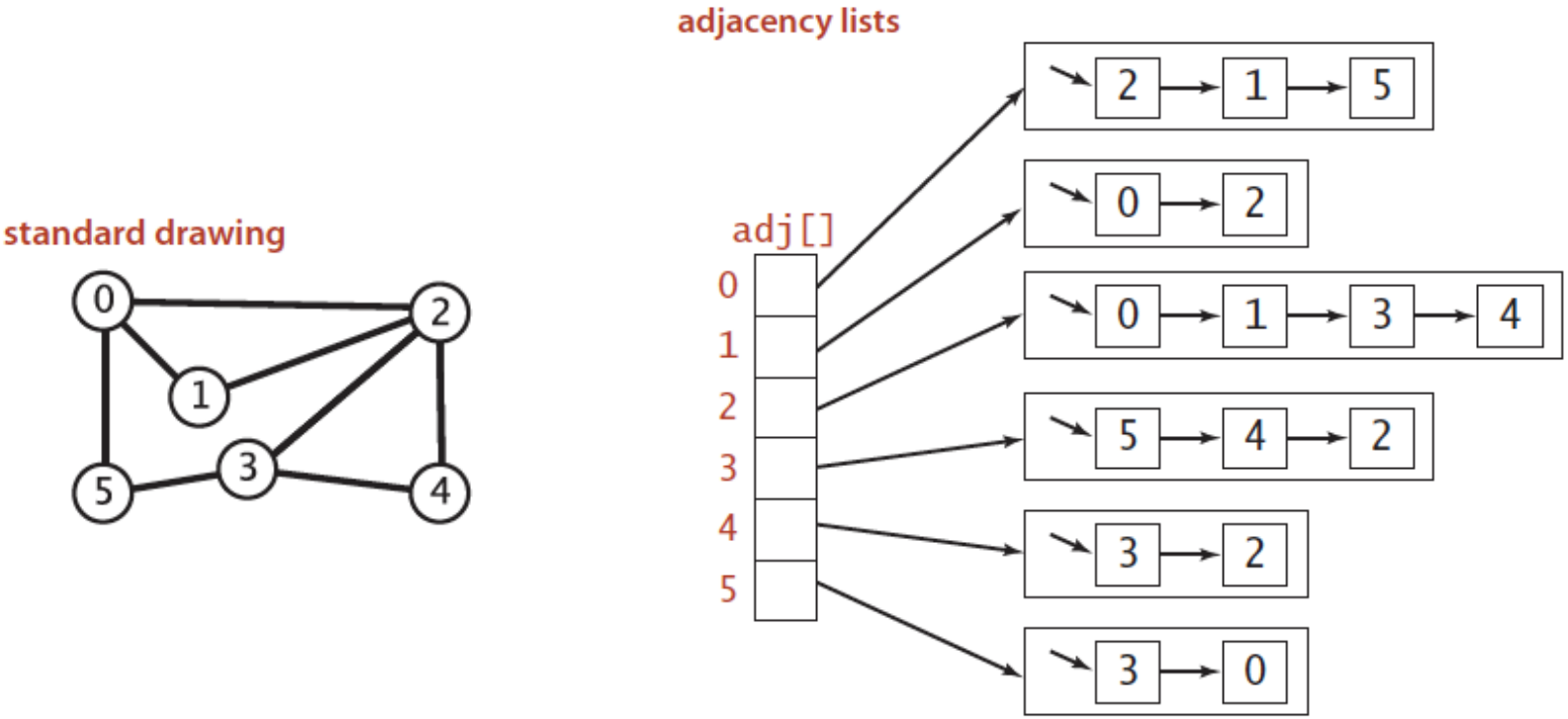
Mark v as visited.

Recursively visit all unmarked vertices w adjacent to v.

Typical applications.

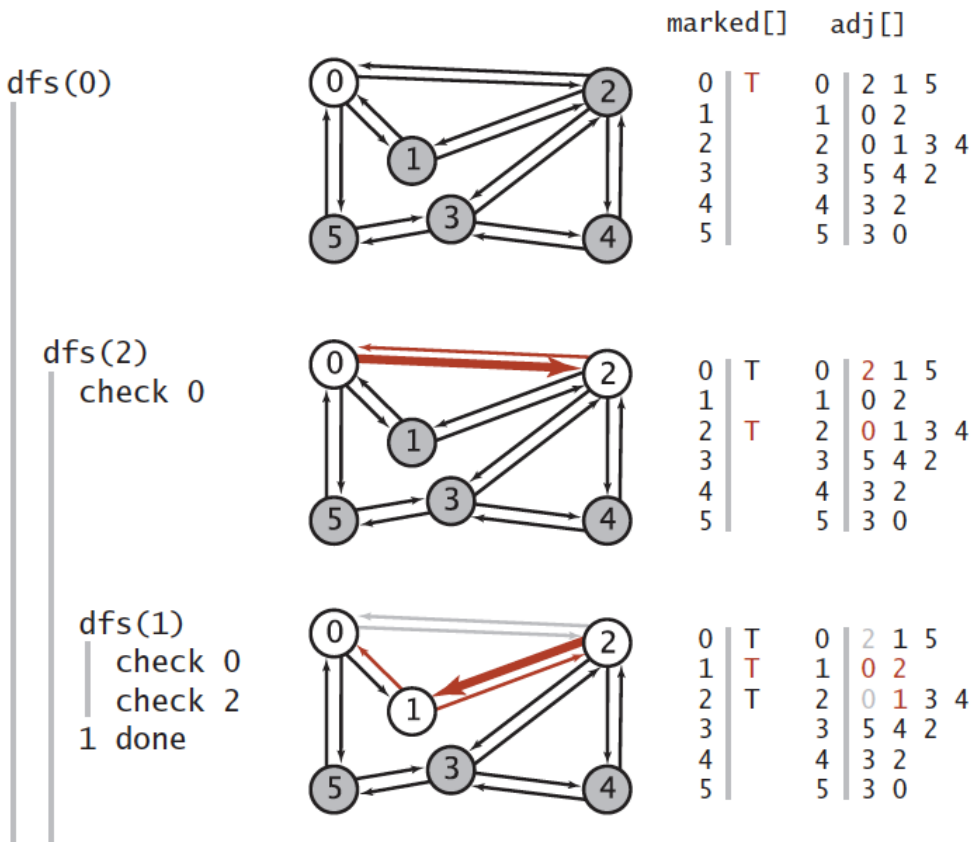
- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Depth First Search Example - I



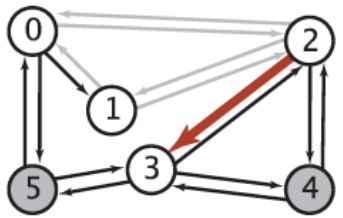
Depth First Search Example - II

DFS explores edges out of the most recently discovered vertex v that still has unexplored leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.



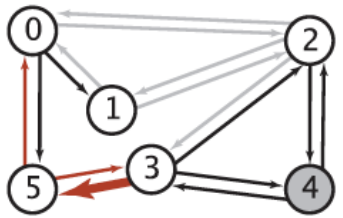
Depth First Search Example - III

```
dfs(3)
  dfs(5)
    check 3
    check 0
  5 done
  dfs(4)
    check 3
    check 2
  4 done
  check 2
3 done
check 4
2 done
check 1
check 5
0 done
```



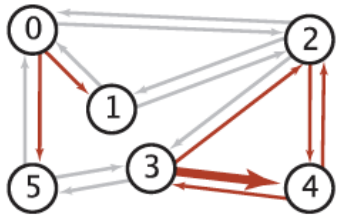
0	T
1	T
2	T
3	T
4	
5	

0	2 1 5
1	0 2
2	0 1 3 4
3	5 4 2
4	3 2
5	3 0



0	T
1	T
2	T
3	T
4	
5	T

0	2 1 5
1	0 2
2	0 1 3 4
3	5 4 2
4	3 2
5	3 0



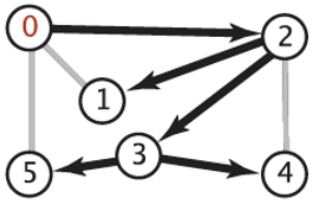
0	T
1	T
2	T
3	T
4	T
5	T

0	2 1 5
1	0 2
2	0 1 3 4
3	5 4 2
4	3 2
5	3 0

DFS results

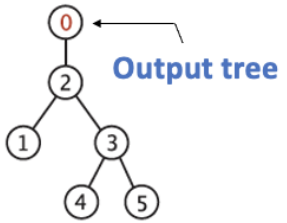
- To save known paths to each vertex the dfs() function in the textbook, maintains a vertex-indexed array edgeTo[] such that edgeTo[w] = v means that v-w was the edge used to access w for the first time.
- The edgeTo[] array is a parent-link representation of a tree rooted at s that contains all the vertices connected to s. We term this tree as the output tree.

```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w])
      {
         edgeTo[w] = v;
         dfs(G, w);
      }
```



edgeTo[]

0	
1	2
2	0
3	2
4	3
5	3



DFS Example

Question: Draw the output tree for the input undirected graph below, when traversed using DFS with 0 as the source.

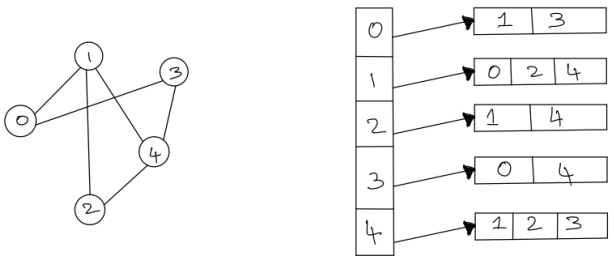


Figure 1: Input Undirected graph
Graph

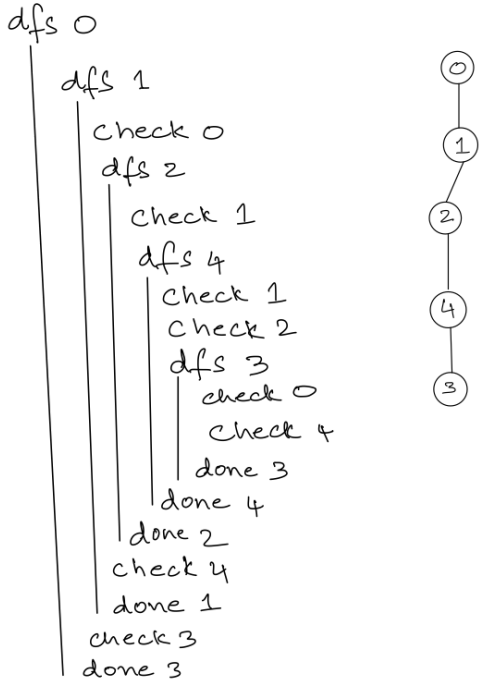


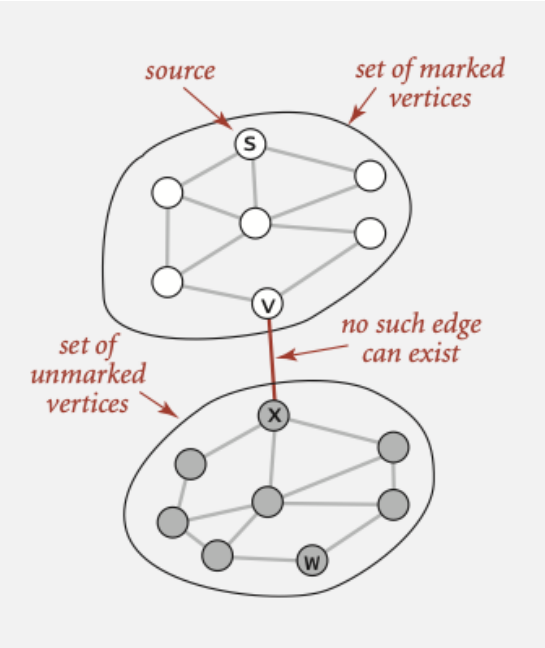
Figure 2: Outline of DFS execution and the output tree

Depth-first search: properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees (plus time to initialize the $marked[]$ array).

Pf. [correctness] First, we prove that the algorithm marks all the vertices connected to the source s (and no others).

- Every marked vertex is connected to s - why?
- Suppose that some unmarked vertex x is connected to s. Since s itself is marked, any path from s to x must have at least one edge from the set of marked vertices to the set of unmarked vertices, say $v \rightarrow x$. But the algorithm would have discovered x after marking v, so no such edge can exist – a contradiction.
- Marking ensures that each vertex is visited once (taking time proportional to its degree to check marks).



Single-source shortest paths. Given a graph and a source vertex s, support queries of the form: Is there a path from s to a given target vertex v? If so, find a shortest such path (one with a minimal number of edges).

DFS offers little assistance in solving this problem, because the order in which it takes us through the graph has no relationship to the goal of finding shortest paths.

The classical method for accomplishing this task, called breadth-first search (BFS).

Breadth-first search

BFS (from source vertex s)

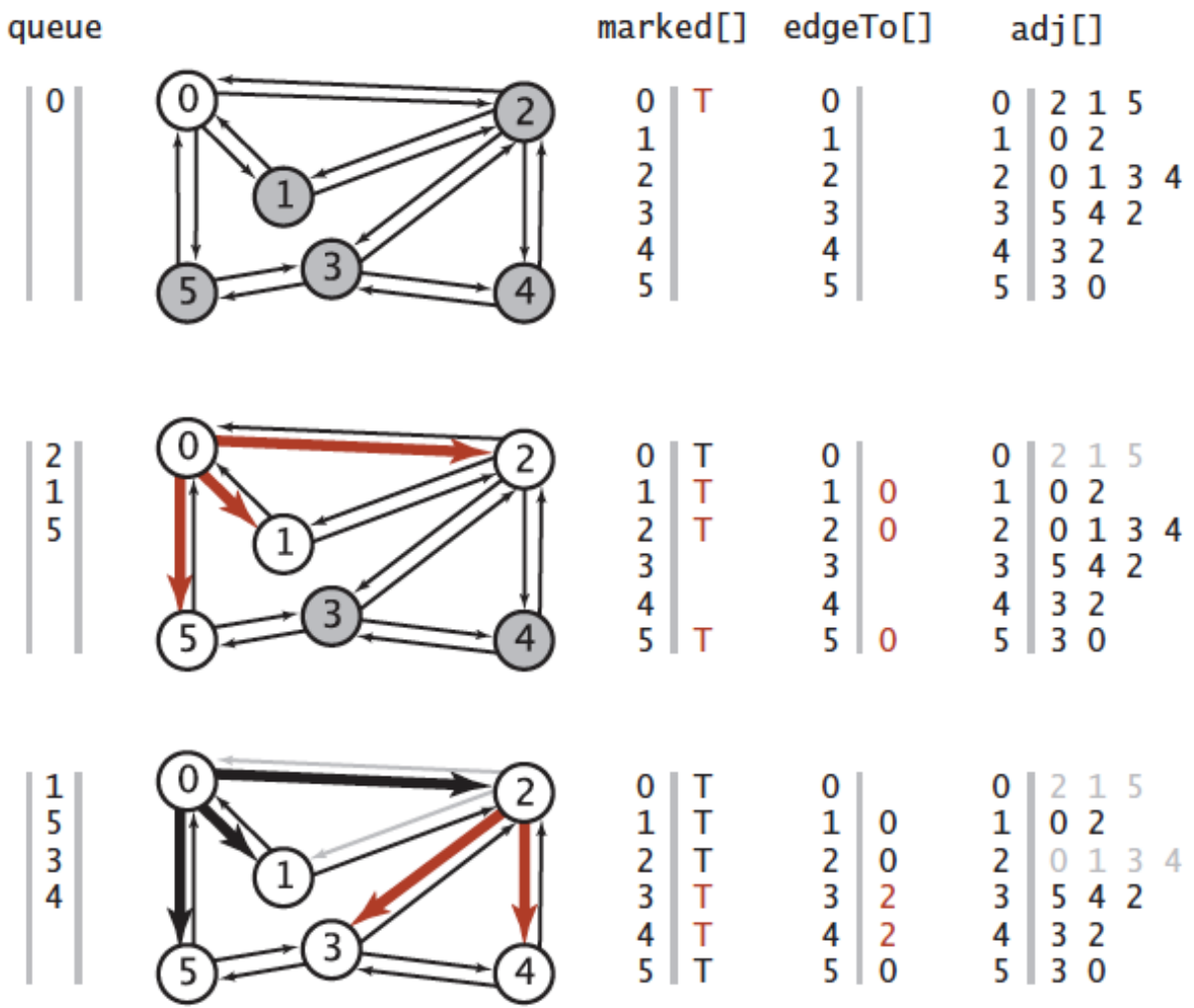
Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- **remove the least recently added vertex v**
- **add each of v's unvisited neighbors to the queue, and mark them as visited.**

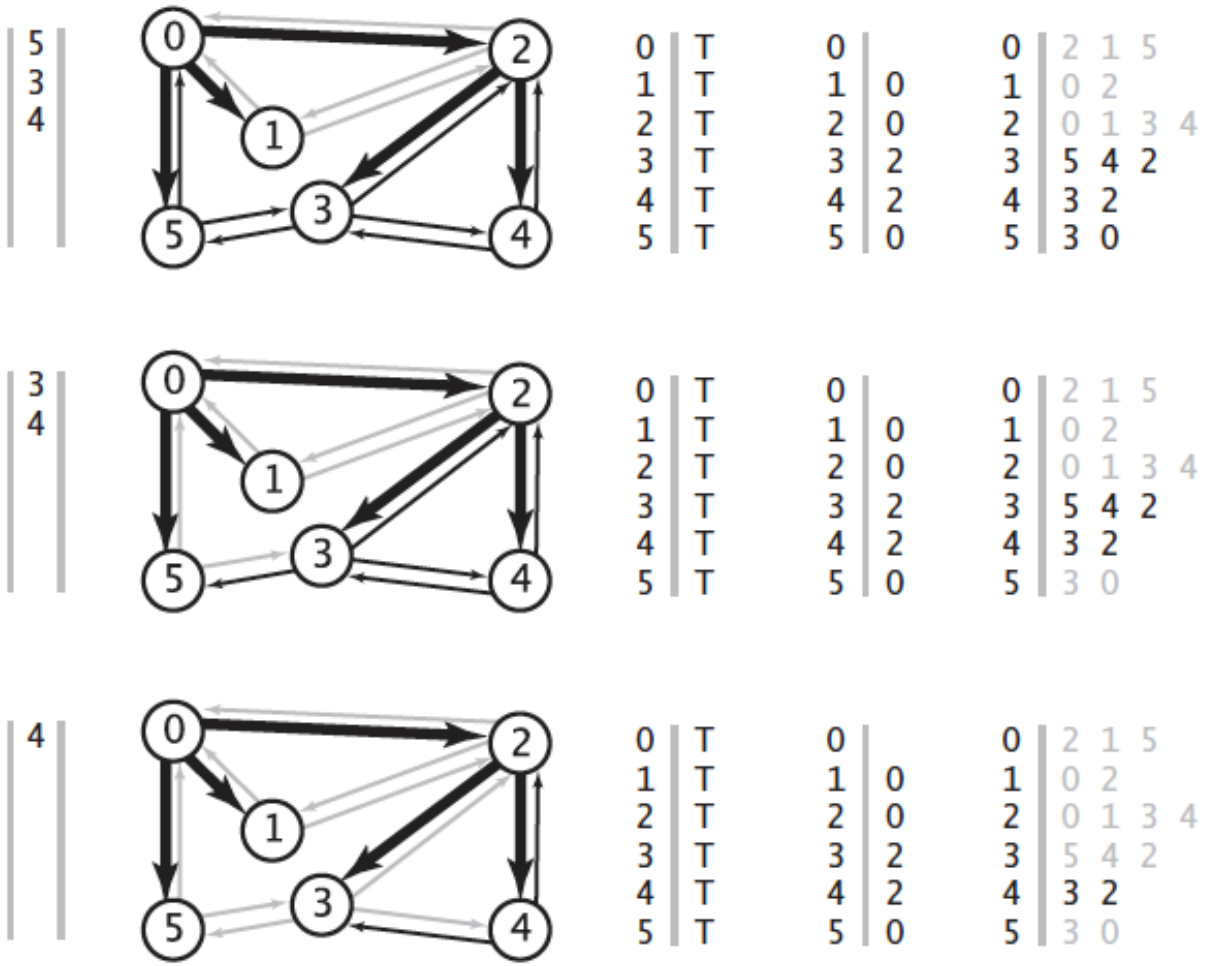
Breadth-first search: Example I

Trace of breadth-first search to find all paths from 0



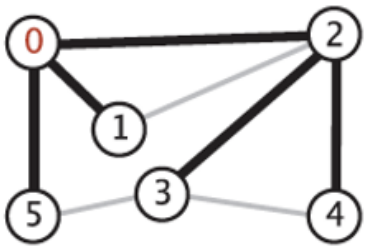
Breadth-first search: Example II

Trace of breadth-first search to find all paths from 0



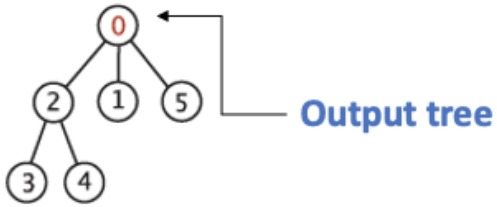
Breadth-first search: code and output tree

```
private void bfs(Graph G, int s)
{
   Queue<Integer> queue = new Queue<Integer>();
   marked[s] = true;           // Mark the source
   queue.enqueue(s);           //   and put it on the queue.
   while (!q.isEmpty())
   {
      int v = queue.dequeue(); // Remove next vertex from the queue.
      for (int w : G.adj(v))
         if (!marked[w])       // For every unmarked adjacent vertex,
         {
            edgeTo[w] = v;     //   save last edge on a shortest path,
            marked[w] = true;  //   mark it because path is known,
            queue.enqueue(w);  //   and add it to the queue.
         }
   }
```



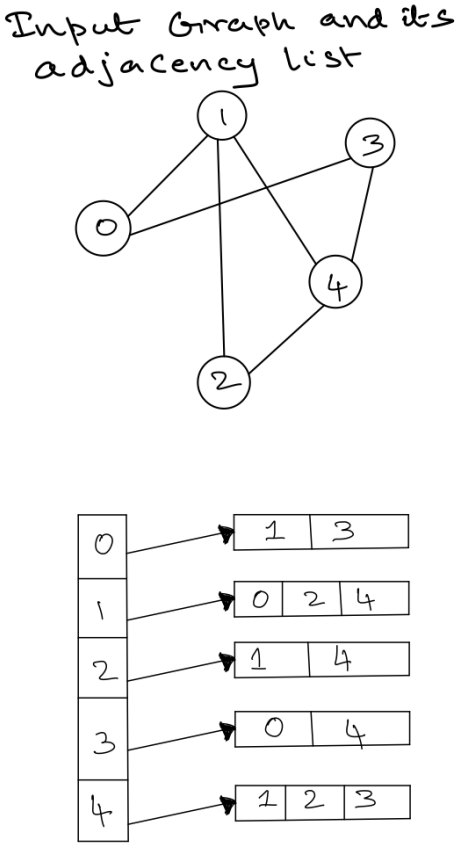
edgeTo[]

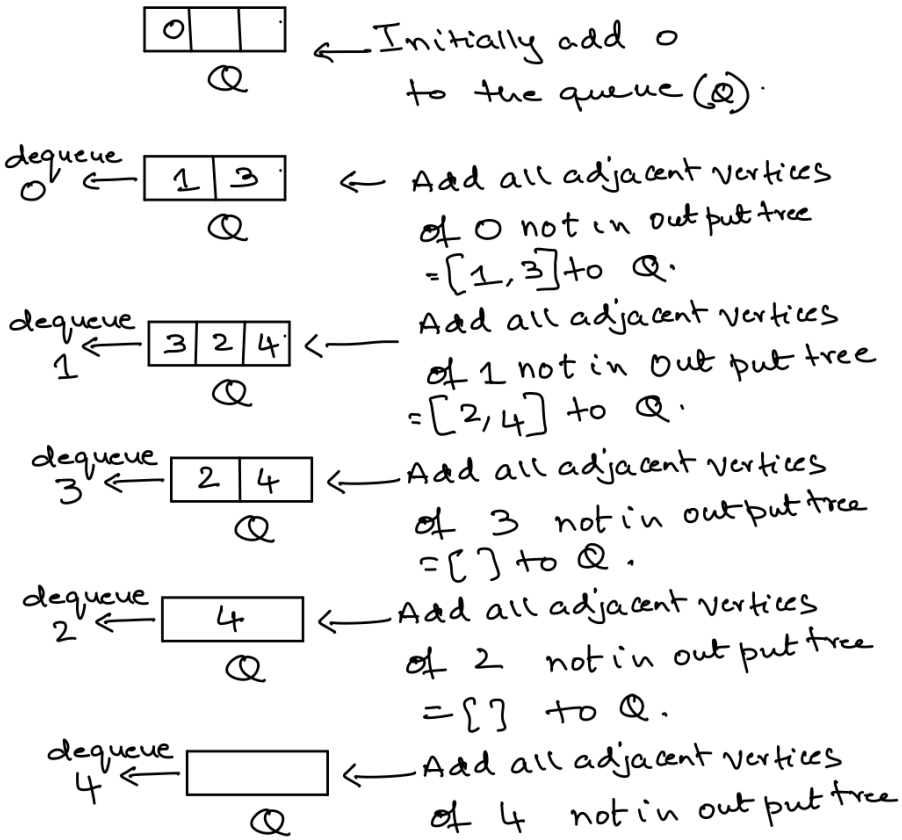
0	
1	0
2	0
3	2
4	2
5	0

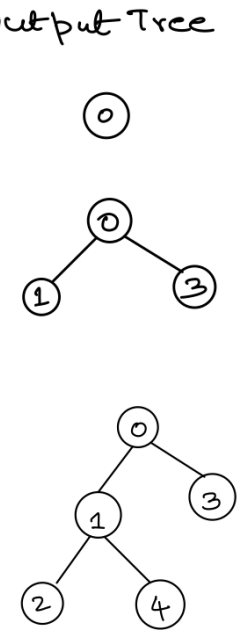


Outcome of breadth-first search to find all paths from 0

Breadth-first search: Example





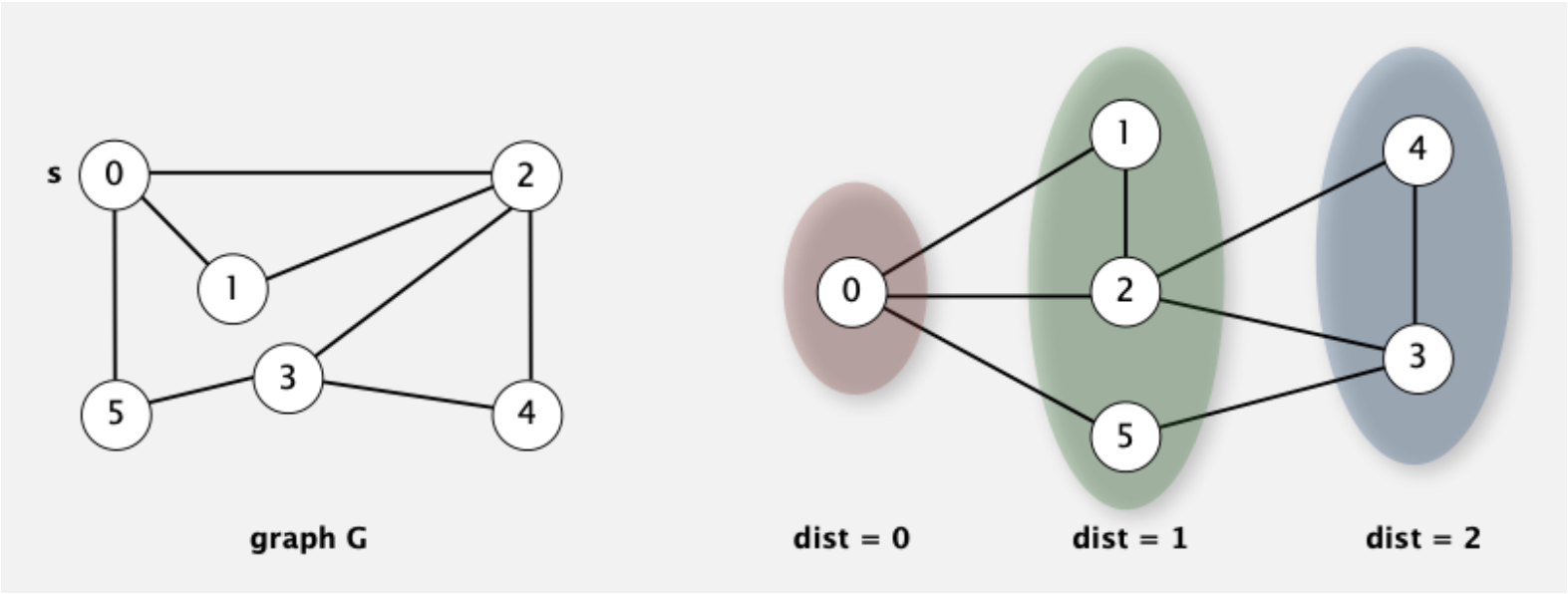


Breadth-first search: Properties

Q. In which order does BFS examine vertices?

A. Increasing distance (number of edges) from s. [queue always consists of ≥ 0 vertices of distance k from s, followed by ≥ 0 vertices of distance $k+1$]

Proposition. In any connected graph G, BFS computes shortest paths from s to all other vertices in time proportional to $E + V$.



DFS and BFS

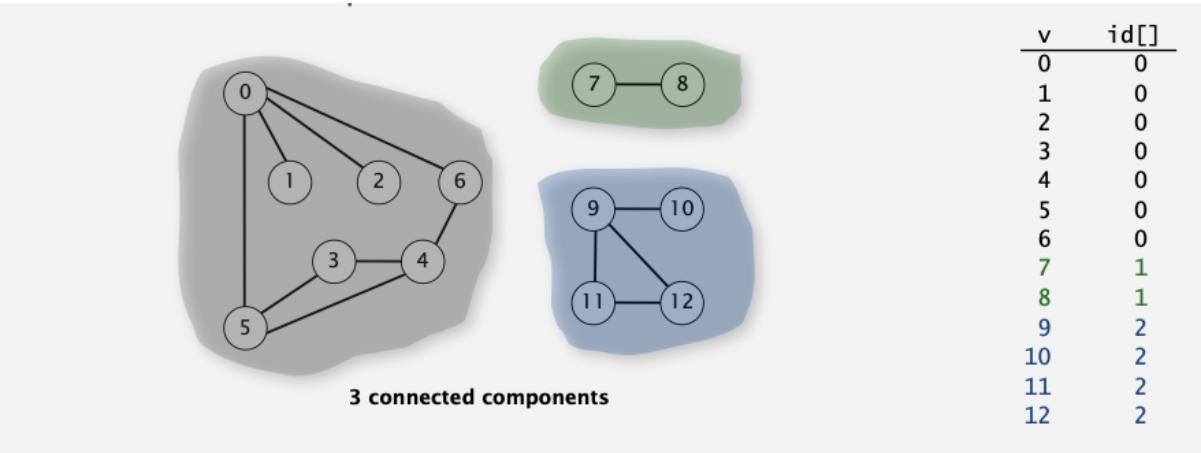
- DFS and BFS are fundamental algorithms for searching graphs.
- In both, we put the source vertex on the data structure, then perform the following steps until the data structure is empty:
  - Take the next vertex v from the data structure and mark it.
  - Put onto the data structure all unmarked vertices that are adjacent to v.
- The algorithms differ only in the rule used to take the next vertex from the data structure (least recently added for BFS, most recently added for DFS).
- This difference leads to completely different views of the graph, even though all the vertices and edges connected to the source are examined no matter what rule is used.

Connected components

Recall from Chapter 1, the relation "is connected to" is an equivalence relation:

- Reflexive: v is connected to v.
- Symmetric: if v is connected to w, then w is connected to v.
- Transitive: if v is connected to w and w is connected to x, the v is connected to x.

Def. A connected component is a maximal set of connected vertices.



Remark. Given connected components, can answer queries in constant time.

Connected components: Depth First Search

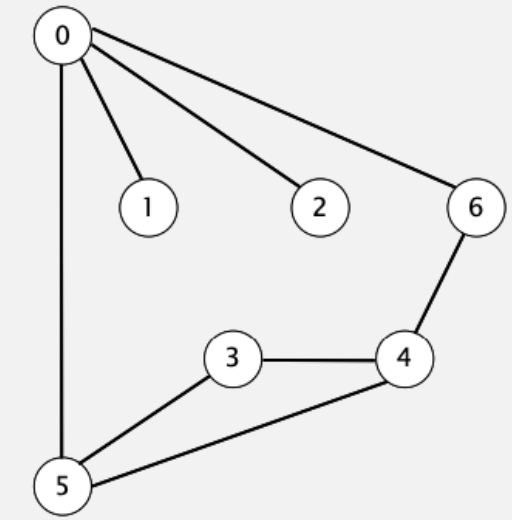
Goal. Partition vertices into connected components.

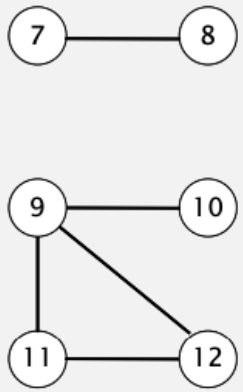
Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.

Connected components: DFS Example I



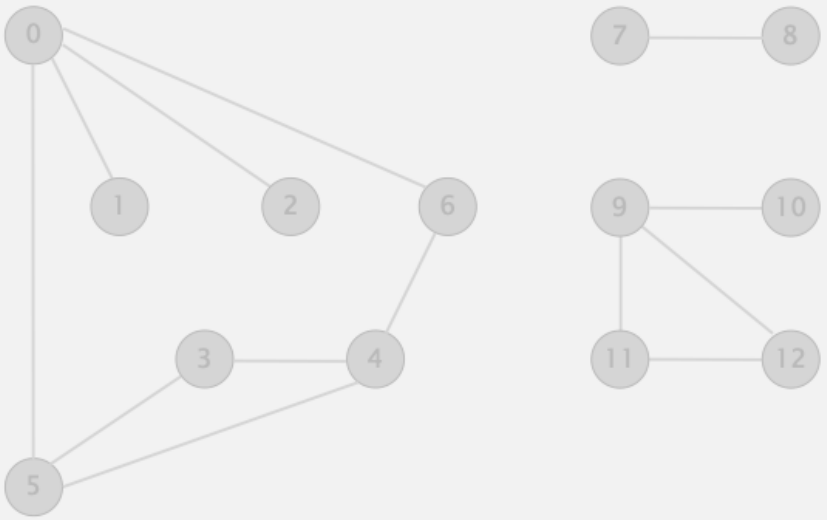


v	marked[]	id[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

graph G

See Demo: `https://algs4.cs.princeton.edu/lectures/`

Connected components: DFS Example II



v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

done

See Demo: `https://algs4.cs.princeton.edu/lectures/`

Connected components: sample code

```
public CC(Graph G)
{
   marked = new boolean[G.V()];
   id = new int[G.V()];
   for (int s = 0; s < G.V(); s++)
      if (!marked[s])
      {
          dfs(G, s);
          count++;
      }
}

private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
          dfs(G, w);
}
```

Connected components: DFS analysis

Proposition. DFS using preprocessing space and time proportional to (V+E) support constant-time connectivity queries in a graph.

Proof: Immediate from the code. Each adjacency-list entry is examined exactly once, and there are 2E such entries (two for each edge). Instance methods examine or return one or two instance variables.

Connected components: Union-Find Vs. DFS

- In theory, DFS is faster than union-find because it provides a constant-time guarantee, which union-find does not.
- In practice, however, this difference is negligible, and union-find is faster because it does not have to build a full representation of the graph.
- More importantly, union-find is an online algorithm (we can check whether two vertices are connected in near-constant time at any point, even while adding edges), whereas the DFS solution must first preprocess the graph.
- Therefore, for example, we prefer union-find when determining connectivity is our only task or when we have a large number of queries intermixed with edge insertions, but may find the DFS solution more appropriate for use in a graph ADT because it makes efficient use of existing infrastructure.